UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

Práctica 2
MPI2
SISTEMA DISTRIBUIDO DE RENDERIZADO DE GRÁFICOS

Carlos Gómez Fernández

Asignatura: Diseño de Infraestructura de Red

Titulación: Grado en Ingeniería Informática

Fecha: 1 de Mayo de 2021

ÍNDICE.

1 ENUNCIADO DEL PROBLEMA.

Se emplearán las primitivas pertinentes *MPI2* como acceso paralelo a disco y gestión de procesos dinámico. Inicialmente el usuario lanzará un único proceso ejecutando *mpirun -np 1 ./pract2*, mediante el cual, *MPI* lanzará un primer proceso que tendrá acceso a la pantalla de gráficos pero no a disco. Este proceso será el encargado de levantar *N* procesos, con *N* definido en tiempo de compilación como una constante, los cuales, tendrán acceso a disco pero no a gráficos directamente.

Los procesos lanzados se encargarán de leer de forma paralela los datos del archivo *foto.dat*, para posteriormente ir enviando los pixeles al primer elemento de proceso para que éste se encargue de representarlo en pantalla.

Se hará uso de la plantilla *pract2.c* para comenzar a desarrollar la práctica. En ella se debe completar el código que ejecuta el proceso con acceso a la ventana de gráficos (rank 0 inicial) y la de los procesos "*trabajadores*".

Se proporciona el archivo *foto.dat*, cuya estructura interna es de 400 filas por 400 columnas de puntos, estando formados por una tripleta de tres *unsigned char*, correspondiente con los valor R, G y B de cada uno de los colores primarios, valores que pueden ser usados en la función *dibujaPunto*.

2 PLANTEAMIENTO DE LA SOLUCIÓN.

2.1 FUNDAMENTOS TEÓRICOS.

El desarrollo de la práctica se enmarca en el renderizado de gráficos, es por ello que será conveniente disponer de un conocimiento previo de la estructura de la imagen a renderizar.

Se puede realizar una abstracción de una imagen, realizando un paralelismo con una estructura comúnmente conocida, como es el caso de una matriz. En el caso concreto que se plantea, se trabaja sobre una matriz de 400 filas de pixeles por 400 columnas de píxeles, es decir, la imagen sobre la que se trabajará consta de un total de 160.000 pixeles.

2.2 FLUJO GENERAL DE EVENTOS.

El programa comenzará con la creación de un único proceso correspondiente con el *rank* 0, el cual creará una ventana, mediante la función *initX*, empleada para la representación del archivo gráfico, haciendo uso de la librería *lX11*. El proceso con *rank* 0 llevará a cabo la creación de *N* procesos "trabajadores", siendo *N*, una constante definida en tiempo de compilación por el usuario.

El proceso con *rank* 0 esperará que los procesos "trabajadores" envíen los pixeles sobre los que trabajaron, para posteriormente ser él mismo quien los muestre en la ventana gráfica generada anteriormente. Previo a la recepción de los pixeles por parte del proceso con *rank* 0, a cada proceso "trabajador" se le asignará una región del archivo gráfico a procesar, la cual se obtendrá de dividir el número de filas de la imagen, entre el total de procesos "trabajadores" generador, asignando las filas sobrantes, al último de los "trabajadores", es decir, al que cuyo *rank* sea *N-1*.

3 DISEÑO DEL PROGRAMA.

3.1 FUNCIONALIDADES DEL PROCESO CON RANK 0.

El proceso con *rank* 0 corresponde con el proceso principal, entre cuyas funciones se encuentra la creación de la ventana gráfica en la que mostrar el archivo gráfico, la creación de los procesos “trabajadores” y la recepción de cada uno de los pixeles procesados por los procesos “trabajadores”, para posteriormente mostrarlos en la ventana gráfica generada.

3.2 FUNCIONALIDADES DEL RESTO DE PROCESOS.

Los procesos “trabajadores” calcularán la región con la trabajarán, es decir desde que fila, hasta que fila renderizarán, consiguiendo así que todos los procesos “trabajadores” accedan concurrentemente al mismo archivo gráfico, obtendrán la información almacenada en cada uno de los pixeles, procesarán la información relacionada con el color del pixel, aplicarán uno de los filtros disponibles y enviarán de nuevo la información modificada asociada a cada pixel al proceso con *rank* 0.

3.3 DISEÑO DE LA SOLUCIÓN.

3.3.1 FUNCIONES PRINCIPALES.

- Función *main*: función principal del programa en la que se realiza una diferenciación entre la funcionalidad correspondiente con el proceso cuyo *rank* es 0 y la de los procesos “trabajadores”.

- Función *recibirPunto*: función ejecutada por el proceso cuyo *rank* es 0, mediante la cual se recibe la información asociada a cada uno de los pixeles del archivo gráfico, enviada por los procesos “trabajadores”, almacenándola en un vector de enteros de cinco posiciones, para posteriormente mostrarla en la ventana gráfica generada con la función *initX*, mediante la función *dibujaPunto*.

- Función *distribuirFilas*: función ejecutada por los procesos “trabajadores”, mediante la cual se calcula la zona de trabajo asociada a cada uno de los procesos “trabajadores”, es decir, las filas que le corresponden a cada “trabajador”, las líneas sobrantes, las cuales serán asignadas al último “trabajador” generado, y el área de trabajo de cada “trabajador”.

- Función *asignarFilas*: función ejecutada por los procesos “trabajadores”, mediante la cual se asigna la zona de trabajo a cada proceso “trabajador”, es decir, se asigna la fila desde la que debe empezar a trabajar, hasta la fila que debe renderizar.

- Función *aperturaFoto*: función ejecutada por los procesos “trabajadores”, mediante la cual se lleva a cabo la apertura del archivo gráfico y el posicionamiento de cada uno de los procesos “trabajadores” en el inicio de su zona de trabajo.

- Función *obtenerPunto*: función ejecutada por los procesos “trabajadores”, mediante la cual se obtendrá de cada pixel procesado por cada “trabajador”, la información asociada a los colores y se almacenará en un vector auxiliar de *unsigned char* de tres posiciones, sobre el cual se aplicará el filtro, al realizar la llamada a la función *aplicarFiltro*.

3.3.2 FUNCIONES AUXILIARES

- Función *aplicarFiltro*: función ejecutada por los procesos “trabajadores”, mediante la cual se aplica el filtro a cada pixel del archivo gráfico, pudiendo ser cuatro tipo de filtros: la imagen por defecto, es decir, sin aplicar ningún filtro, un filtro en escala de grises, un filtro sepia y un filtro en negativo de la imagen. Tras aplicar alguno de estos filtros, se comprobará que se aplicó correctamente mediante la función *comprobarPunto*, para finalmente enviar la información actualizada asociada a cada pixel al proceso cuyo *rank* es 0, encargado de mostrar el resultado.

- Función *comprobarPunto*: función ejecutada por los procesos “trabajadores”, mediante la cual se comprueba que el valor asociado al color actualizado del pixel se encuentra en el rango adecuado, el cual se encuentra entre 0 y 255.

4 FUENTES DEL PROGRAMA.

4.1 PRACT2.C.

```
/* Pract2  RAP 09/10    Javier Ayllon */
/* SISTEMA DISTRIBUIDO DE RENDERIZADO DE GRAFICOS */
/* Carlos Gomez Fernandez */

#include <openmpi/mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <X11/Xlib.h>
#include <assert.h>
#include <unistd.h>

#define NIL (0)
//#define N_TRABAJADORES 5
//#define FILTRO 0

/* Archivo grafico a renderizar */
#define IMAGEN "foto.dat"
/* Cantidad total de filas que componen el archivo grafico a procesar */
#define N_FILAS 400
/* Cantidad total de columnas que componen el archivo grafico a procesar */
#define N_COLUMNAS 400
/* Dimensiones del archivo grafico a procesar */
#define TAMANIO_FOTO N_FILAS*N_COLUMNAS
/* Desplazamiento en el eje X, a traves de las columnas */
#define X 0
/* Desplazamiento en el eje Y, a traves de las filas */
#define Y 1
/* Color primario rojo */
#define R 2
/* Color primario verde */
#define G 3
/* Color primario azul */
#define B 4
```

```
/* Informacion asociada a cada pixel */
#define PIXEL 5

/* Variables Globales */
XColor colorX;
Colormap mapacolor;
char cadenaColor[]="#000000";
Display *dpy;
Window w;
GC gc;

/* Funciones principales empleadas para la resolucion del problema */
void initX();
void dibujaPunto(int x, int y, int r, int g, int b);
void recibirPunto(MPI_Comm commPadre);
void distribuirFilas(int *filasTrabajador, int *filasNoAsignadas, int *areaTrabaja-
dor);
void asignarFilas(int rank, int *filaInicial, int *filaFinal, int filasTrabaja-
dor, int filasNoAsignadas);
MPI_File aperturaFoto(int rank, int areaTrabajador);
void obtenerPunto(int rank, int filaInicial, int filaFi-
nal, MPI_File foto, MPI_Comm commPadre);

/* Funciones auxiliares empleadas para la resolucion del problema */
void aplicarFiltro(int fila, int columna, unsigned char *colorPunto, MPI_Comm comm-
Padre);
void comprobarPunto(int *bufPunto);

/* Programa principal */
int main(int argc, char *argv[]){
      int rank, size, codigosError[N_TRABAJADORES];
      MPI_Comm commPadre;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);
      MPI_Comm_get_parent(&commPadre);

      if((commPadre==MPI_COMM_NULL) && (rank==0)){
            /* Codigo del maestro */
            /* Funcion para inicializar la ventana en la que se mostrara el ar-
chivo grafico */
            initX();
            /* Creacion de los trabajadores y asociacion del programa a ejecu-
tar a cada uno de ellos */
            MPI_Comm_spawn("pract2", MPI_ARGV_NULL, N_TRABAJADO-
RES, MPI_INFO_NULL, 0, MPI_COMM_WORLD, &commPadre, codigosError);
```

```
            /* Funcion para recibir la informacion asociada a cada pixel y mos-
trar el archivo grafico por pantalla */
            recibirPunto(commPadre);
            /* Espera para mostrar el archivo grafico renderizado */
            sleep(4);
      }else{
            /* Codigo de todos los trabajadores */
            /* El archivo sobre el que debemos trabajar es foto.dat */

            int filasTrabajador, filasNoAsignadas, areaTrabajador, filaInicial, fi-
laFinal;
            /* Manejador del archivo grafico */
            MPI_File foto;
            /* Funcion para calcular la carga de trabajo de los trabajadores gene-
rados, para realizar el renderizado del archivo grafico */
            distribuirFilas(&filasTrabajador, &filasNoAsignadas, &areaTrabajador);
            /* Funcion para distribuir la carga de trabajo entre los trabajado-
res generados, para realizar el renderizado del archivo grafico */
            asignarFilas(rank, &filaInicial, &filaFinal, filasTrabajador, fi-
lasNoAsignadas);
            /* Funcion para abrir el archivo grafico */
            foto = aperturaFoto(rank, areaTrabajador);
            /* Funcion para obtener la informacion, respecto del color, aso-
ciada a cada pixel */
            obtenerPunto(rank, filaInicial, filaFinal, foto, commPadre);
            /* Cierre del archivo grafico */
            MPI_File_close(&foto);
            /* Finalizacion de la ejecucion */
            MPI_Finalize();
            return 0;
      }
}

/* Funcion para inicializar la ventana en la que se mostrara el archivo grafico */
void initX(){

      dpy = XOpenDisplay(NIL);
      assert(dpy);

      int blackColor = BlackPixel(dpy, DefaultScreen(dpy));
      int whiteColor = WhitePixel(dpy, DefaultScreen(dpy));

      w = XCreateSimpleWindow(dpy, DefaultRootWindow(dpy), 0, 0, 400, 400, 0, bla-
ckColor, blackColor);
      XSelectInput(dpy, w, StructureNotifyMask);
      XMapWindow(dpy, w);
      gc = XCreateGC(dpy, w, 0, NIL);
      XSetForeground(dpy, gc, whiteColor);
```

```
	for(;;){
		XEvent e;
		XNextEvent(dpy, &e);
		if(e.type == MapNotify){
			break;
		}
	}

	mapacolor = DefaultColormap(dpy, 0);
}

/* Funcion para mostrar por pantalla cada uno de los pixeles que componen el archivo grafico */
void dibujaPunto(int x, int y, int r, int g, int b){

	sprintf(cadenaColor, "#%.2X%.2X%.2X", r, g, b);
	XParseColor(dpy, mapacolor, cadenaColor, &colorX);
	XAllocColor(dpy, mapacolor, &colorX);
	XSetForeground(dpy, gc, colorX.pixel);
	XDrawPoint(dpy, w, gc, x, y);
	XFlush(dpy);
}

/* Funcion para recibir la informacion asociada a cada pixel y mostrar el archivo grafico por pantalla */
void recibirPunto(MPI_Comm commPadre){
	/* Vector de enteros, con cinco posiciones, que almacena la informacion asociada a cada pixel */
	int bufPunto[PIXEL];
	/* Bucle para la recepcion de la informacion asociada a cada pixel, con tantas iteraciones como pixeles disponga el archivo grafico, para su posterior muestra */
	for(int i = 0; i < TAMANIO_FOTO; i++){
		/* Recepcion de cada uno de los pixeles que conforman el archivo grafico */
		MPI_Recv(&bufPunto, PIXEL, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, commPadre, MPI_STATUSES_IGNORE);
		/* En algun momento dibujamos puntos en la ventana algo como dibujaPunto(x,y,r,g,b); */
		dibujaPunto(bufPunto[X],bufPunto[Y],bufPunto[R],bufPunto[G],bufPunto[B]);
	}
}
```

```
/* Funcion para calcular la carga de trabajo de los trabajadores genera-
dos, para realizar el renderizado del archivo grafico */
void distribuirFilas(int *filasTrabajador, int *filasNoAsignadas, int *areaTrabaja-
dor){
	/* Calcular las filas que procesara cada trabajador */
	*filasTrabajador = N_FILAS/N_TRABAJADORES;
	/* Calcular las filas no asignadas a ningun trabajador, las cuales serán pro-
cesadas por el último trabajador */
	*filasNoAsignadas = N_FILAS%N_TRABAJADORES;
	/* Calcular el area que procesara cada trabajador */
	*areaTrabajador = *filasTrabajador*N_COLUMNAS*3*sizeof(unsigned char);
}

/* Funcion para distribuir la carga de trabajo entre los trabajadores genera-
dos, para realizar el renderizado del archivo grafico */
void asignarFilas(int rank, int *filaInicial, int *filaFinal, int filasTrabaja-
dor, int filasNoAsignadas){
	/* Calcular la fila desde la que comenzaran a trabajar los trabajadores gene-
rados, para realizar el renderizado del archivo grafico */
	*filaInicial = rank*filasTrabajador;
	/* Calcular la fila hasta la que trabajaran los trabajadores genera-
dos, para realizar el renderizado del archivo grafico */
	if(rank != N_TRABAJADORES-1){
		/* Fila hasta la que trabajaran los trabajadores generados, para reali-
zar el renderizado del archivo grafico */
		*filaFinal = (rank+1)*filasTrabajador;
	}else{
		/* Fila hasta la que trabajara el ultimo de los trabajadores genera-
dos, para realizar el renderizado del archivo grafico */
		*filaFinal = (rank+1)*filasTrabajador+filasNoAsignadas;
	}
}

/* Funcion para abrir el archivo grafico */
MPI_File aperturaFoto(int rank, int areaTrabajador){
	/* Manejador del archivo grafico */
	MPI_File foto;
	/* Apertura, por parte de todos los procesos, del archivo gra-
fico "foto.dat" en modo solo lectura */
	MPI_File_open(MPI_COMM_WORLD, IMAGEN, MPI_MODE_RDONLY, MPI_INFO_NULL, &foto);
	/* Asignar el area del archivo grafico visible a cada trabajador generado */
	MPI_File_set_view(foto, rank*areaTrabajador, MPI_UNSIGNED_CHAR, MPI_UNSIG-
NED_CHAR, "native", MPI_INFO_NULL);

	return foto;
}
```

```
/* Funcion para obtener la informacion, respecto del color, asociada a cada pi-
xel */
void obtenerPunto(int rank, int filaInicial, int filaFi-
nal, MPI_File foto, MPI_Comm commPadre){
	/* Vector de chars, con tres posiciones, que almacena la informacion, res-
pecto del color, asociada a cada pixel */
	char colorPunto[3];
	/* Bucle para la recepcion de la informacion, respecto del color de cada pi-
xel, con tantas iteraciones como pixeles disponga el archivo grafico, para su pos-
terior muestra */
	for(int i = filaInicial; i<filaFinal; i++){
		for(int j = 0; j<N_COLUMNAS; j++){
			/* Lectura y almacenamiento de los valores, respecto del co-
lor, asociado a cada uno de los pixeles */
			MPI_File_read(foto, colorPunto, 3, MPI_UNSIGNED_CHAR, MPI_STA-
TUS_IGNORE);
			/* Funcion para aplicar un filtro a cada pixel del archivo gra-
fico */
			aplicarFiltro(i, j, colorPunto, commPadre);
		}
	}
}

/* Funcion para aplicar un filtro a cada pixel del archivo grafico */
void aplicarFiltro(int fila, int columna,  unsigned char *colorPunto, MPI_Comm com-
mPadre){
	/* Vector de enteros, con cinco posiciones, que almacena la informacion aso-
ciada a cada pixel */
	int bufPunto[PIXEL];
	/* Valor asociado al desplazamiento en el eje X, a traves de las columnas */
	bufPunto[X] = columna;
	/* Valor asociado al desplazamiento en el eje Y, a traves de las columnas */
	bufPunto[Y] = fila;
	/* Seleccion del filtro a aplicar en  */
	switch(FILTRO){
		/* Imagen original */
		case 0:
			bufPunto[R] = ((int)colorPunto[0]);
			bufPunto[G] = ((int)colorPunto[1]);
			bufPunto[B] = ((int)colorPunto[2]);
			break;
```

```
            /* Filtro escala de grises a aplicar sobre el archivo grafico */
            case 1:
                  bufPunto[R] = (((int)colorPunto[0])*0.299)+(((int)color-
Punto[1])*0.587)+(((int)colorPunto[2])*0.114);
                  bufPunto[G] = (((int)colorPunto[0])*0.299)+(((int)color-
Punto[1])*0.587)+(((int)colorPunto[2])*0.114);
                  bufPunto[B] = (((int)colorPunto[0])*0.299)+(((int)color-
Punto[1])*0.587)+(((int)colorPunto[2])*0.114);
                  break;
            /* Filtro sepia a aplicar sobre el archivo grafico */
            case 2:
                  bufPunto[R] = (((int)colorPunto[0])*0.393)+(((int)color-
Punto[1])*0.769)+(((int)colorPunto[2])*0.189);
                  bufPunto[G] = (((int)colorPunto[0])*0.349)+(((int)color-
Punto[1])*0.686)+(((int)colorPunto[2])*0.168);
                  bufPunto[B] = (((int)colorPunto[0])*0.272)+(((int)color-
Punto[1])*0.534)+(((int)colorPunto[2])*0.131);
                  break;
            /* Filtro negativo a aplicar sobre el archivo grafico */
            case 3:
                  bufPunto[R] = 255-((int)colorPunto[0]);
                  bufPunto[G] = 255-((int)colorPunto[1]);
                  bufPunto[B] = 255-((int)colorPunto[2]);
                  break;
      }
      /* Funcion para comprobar que el valor tras aplicar un filtro es va-
lido, es decir, que se encuentra en el rango de 0 a 255 */
      comprobarPunto(bufPunto);
      /* Envio del pixel con el filtro aplicado al maestro */
      MPI_Send(&bufPunto, PIXEL, MPI_INT, 0, 1, commPadre);
}

/* Funcion para comprobar que el valor tras aplicar un filtro es valido, es de-
cir, que se encuentra en el rango de 0 a 255 */
void comprobarPunto(int *bufPunto){
      /* Comprobacion del valor rojo del pixel */
      if(bufPunto[R]>255){
            bufPunto[R]=255;
      }
      /* Comprobacion del valor verde del pixel */
      if(bufPunto[G]>255){
            bufPunto[G]=255;
      }
      /* Comprobacion del valor azul del pixel */
      if(bufPunto[B]>255){
            bufPunto[B]=255;
      }
}
```

5 INSTRUCCIONES COMPILACIÓN Y EJECUCIÓN.

Para llevar a cabo la compilación y ejecución del programa se dispone de un archivo *Makefile*.

Para compilar el programa pract2.*c* sin aplicar ningún filtro al archivo gráfico *foto.dat*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados el archivo fuente, el archivo gráfico y el *Makefile*.
$ make compilarSinFiltro

Para compilar el programa pract2.*c* aplicando un filtro de escala de grises al archivo gráfico *foto.dat*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados el archivo fuente, el archivo gráfico y el *Makefile*.
$ make compilarFiltroBlancoYNegro

Para compilar el programa pract2.*c* aplicando un filtro sepia al archivo gráfico *foto.dat*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados el archivo fuente, el archivo gráfico y el *Makefile*.
$ make compilarFiltroSepia

Para compilar el programa pract2.*c* aplicando un filtro negativo al archivo gráfico *foto.dat*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados el archivo fuente, el archivo gráfico y el *Makefile*.
$ make compilarFiltroNegativo

Para ejecutar el programa pract2.*c*, basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados tanto el archivo fuente, como el *Makefile*.
$ make ejecutarRenderizado

Para eliminar el archivo objeto generado en la compilación del programa pract2.*c* basta con ejecutar el siguiente comando en el directorio donde se encuentren almacenados el archivo fuente, el archivos objeto y el *Makefile*.
$ make limpiarDirectorios